JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 2, March–April 2006

Support for Object Oriented Transactions in Timor

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein, **Gisela Menger**, University of Ulm, Germany
Frans Henskens, Michael Hannaford, University of Newcastle, N.S.W., Australia

Abstract

An important aim in the design of the Timor programming language is to provide programmers with features which enable them to build complex systems from components which can be developed in isolation from each other (i.e. without knowledge of each other's existence). The database transaction concept serves as an interesting test case for this objective, since it is a general concept which can be applied to many different applications. The paper discusses those features of Timor which allow this objective to be achieved.

1 INTRODUCTION

The programming language Timor[1] builds on the basic concepts of object orientation but adds further features not commonly found in OO languages, including the separation of types from their implementations and the separation of subtyping from code reuse. It also adds two new kinds of types, known as qualifying types and attribute types, which allow independent components to be defined and implemented for use in a manner similar to the use of adjectives in natural languages. Just as adjectives are modular units which modify and extend the meaning of nouns, so Timor's "adjectival" types allow the behaviour of objects to be modified and extended in a modular fashion. The approach differs fundamentally from the OO subclassing technique for extending classes in that the adjectival types comprise separate units which can be independently applied to objects of different types in a modular way.

The aim of this paper is to illustrate how qualifying types can be used to develop advanced components which can add new "system" functionality to existing systems without the latter's objects having to be modified in any way at all (i.e. neither at the

[1] see www.timor-programming.org

Cite this column as follows: J. Leslie Keedy, K. Espenlaub, C. Heinlein, G. Menger, F. Henskens, M. Hannaford: "Support for Object Oriented Transactions in Timor", in *Journal of Object Technology*, vol. 5, no. 2, March-April 2006, pp. 103-124 http://www.jot.fm/issues/issue_2006_03/article1

programming language source code level nor in terms of the intermediate or machine code produced by the compiler), assuming that the original application system was written in Timor. A knowledge of the basic paper on qualifying types [5] is assumed in the rest of this paper.

Several cases of enhancing functionality in this sense have already been described in earlier papers [4, 5], including the addition of synchronisation, monitoring and protection to objects. In this paper we present more advanced features of Timor that make it possible to develop components which for example can (a) allow copies of objects to be automatically created and updated in parallel with the original objects in a system, and (b) allow an existing system to be transformed into a transaction processing system in the traditional database sense.

In line with the Timor design philosophy, the additional components developed to achieve such aims must be type safe, both in the sense that they are themselves statically type safe and in the sense that an application system into which such components have been integrated remains type safe.

Two examples allow new constructs to be presented and discussed. The first of these (section 2) illustrates how an existing application can be enhanced to allow a replica of each object in an application to be maintained such that this is updated without the knowledge or cooperation of either the client or the target objects involved. The second, more ambitious, example (section 3) describes how an existing application can be transformed into a transaction processing system, again in such a way that existing objects are unaware of the additional functionality (such as creating and maintaining shadow objects, committing and aborting transactions, etc.). Section 4 considers protection and security issues which are raised by these examples, showing that the owner of a system can prevent the additional components from violating the confidentiality and/or integrity of the application objects. Section 5 discusses related work and section 6 concludes the paper.

2 ACCESSING PARALLEL VERSIONS OF OBJECTS

This first example considers the case where an application system consists of a database of objects which can be modified as the application proceeds. The aim is to enhance the application such that whenever an object is modified a replica is automatically modified in the same way (e.g. in a separate part of the disc store) so that in the case of failure there is always a backup version of each object. To keep this first example relatively simple, we assume that new objects are not dynamically created during the execution of the application system[2]. Instead a program is developed which copies each object in turn and acquires for each a Timor reference. This is a straightforward program which need not be further discussed here.

What is now needed is a set of components which can in effect monitor the write activities carried out on the original object database and surreptitiously make the same

[2] In the next example it will be shown how duplicate objects can be dynamically created.

modifications to the replica database. In principle this is easily achieved using Timor qualifiers (qualifying objects) with call-in bracket methods [5].

The type of such components might be defined along the following lines:

```
type ReplicaWriter {
maker:
 init(Handle aReplica);
qualifies any:
 enq bracket op (...) throws DynamicTypeErr;
}
```

where the maker (constructor) of a component receives as a parameter a reference for the replica which it controls. The type also defines a bracket method which is responsible for the actual updating of the replica.

Figure 1 illustrates in general how a bracket method of a qualifier can "catch" method invocations from clients and then execute code, here as a prelude, i.e. before executing a `body` statement (which causes the method originally invoked by the client to be activated), and, after `body` completes, as a postlude.

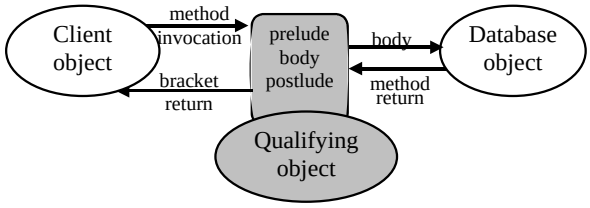


Figure 1: Accessing a qualified Database Object

Such a prelude can contain code which in appropriate circumstances modifies the corresponding replica in the same way as the primary object is to be modified (see Figure 2).

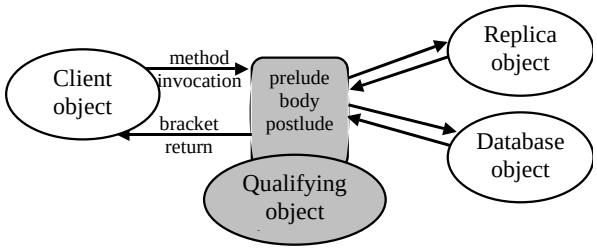


Figure 2: Modifying the Replica Object

The key question is, how can this be achieved in a way which does not require the programmer of the bracket method to know which individual methods of the database object (and its replica) are writer methods, nor what types the database objects have?

In Timor all the instance (and bracket) methods of all types must be categorised in their type definitions either as *operations*, i.e. writer methods (keyword `op`), or as *enquiries*, i.e. reader methods (keyword `enq`). The programmer of a qualifying type can similarly declare bracket methods with the pseudo-identifiers `op` and/or `enq`, i.e. methods for catching writer and/or reader invocations on an object's instance (and bracket) methods[3]. Thus the programmer of a qualifying type does not need a knowledge of the particular types or methods of target objects in order to determine which of their instance methods modify objects.

The next issue is that of the type of the database object and its replica. This can be resolved in that the qualifier receives as a parameter to its own maker a reference for the replica, defined as having the type `Handle` (the supertype of all reference types), so that a reference of *any* type can be passed in this parameter. But how then can the programmer be sure that this is actually the same dynamic type as that of the database object?

A pseudo-type, called `TargetType`, can be used in bracket methods; this represents the dynamic type of the current target object (here the type of the database object). The programmer can use this in a cast statement to ensure that the handle which has been passed for the replica actually has the same dynamic type as the target object, e.g.

```
impl ReplicaWriterImpl of ReplicaWriter {
state:
 Handle theReplica;
maker:
 init(Handle aReplica) {theReplica = aReplica;}
qualifies any:
 enq bracket op (...) throws WrongTargetType {
  cast (theReplica) as {
   (TargetType* myReplica) {/* the code for invoking the
    method of the replica, available here as myReplica */}
  }
  else throw new WrongTargetType.init();
  ...
 }
```

The next question is how the appropriate method of `theReplica` can actually be invoked to update it (without needing a knowledge of its actual type or of the actual method which the client has invoked on the database object which it replicates).

To answer this question we recall that the same problem exists in principle for the bracket method's invocation of the database object itself. In that case the solution is of course provided by the special Timor `body` statement which allows a bracket method to pass on the client's method invocation to its intended target. This mechanism can also be used to invoke other objects of the same dynamic type, such as the replica in the present

[3] Notice that a bracket method which catches `op` methods might itself be declared as an `enq` (because it does not change the state of the qualifier) and vice versa.

example. The redirection of the client's call to the replica is indicated by prefixing to it a variable name (using the normal dot notation)

The only context in which this redirection facility can be used is within a clause of a cast statement which casts to `TargetType`, in order to prevent unexpected run-time type errors. Furthermore, such a clause can only appear within a bracket method, since in Timor the same qualifier can be used concurrently to qualify different target objects; only when a bracket method is active the target – and therefore its type – is actually known. The bracket method can be written as follows:

```
qualifies any:
enq bracket op (...)
                  throws WrongTargetType, ReplicaAccessErr {
 cast (theReplica) as {
  (TargetType* myReplica) {
   try { myReplica.body(...); }
   catch (Exception e) {throw new ReplicaAccessErr.init(e);}
  }
  else throw new WrongTargetType.init();
 }
 return body(...);  // call the database object
}
```

An attempt to invoke the replica directly in the form `theReplica.body(...)` would result in a compile time error, even within the appropriate clause of the cast statement, since the compiler cannot guarantee (either inside or outside the cast clause) that a new object is not assigned to `theReplica`, and such an assignment could result in it having a type other than `TargetType` (because it is declared as a handle). However, the auxiliary variable used within a cast declaration (here `myReplica`) is always implicitly defined to be `fixed` (for a reference) or `final` (for a value).

Finally there remains a protection issue. As the example stands, the writer of the qualifying type could take advantage of the handle passed into its maker to access arbitrary methods of the replica and thus violate the integrity and/or confidentiality of the object. (In order to achieve this a cast statement in which the programmer guesses the type would be necessary, but this possibility cannot be excluded.) We will consider this issue later.

This example has illustrated for a relatively straightforward case how Timor can be used to develop new components which add "system" functionality to an existing application system. In the next section we explore a more ambitious example: enhancing an existing system to transform it into a transaction system.

3 ADDING TRANSACTIONS TO A SYSTEM

The transaction concept is based on the four ACID properties: atomicity, consistency, isolation and durability [2]. The effects of implementing transactions with these properties are that (a) each transaction is regarded as an atomic action in the sense that it either executes to completion or it has no effect on the state of the system, (b) the database is left in a consistent state after the completion of each transaction, (c) transactions which execute in parallel are logically isolated from each other, and (d) after a transaction completes successfully its effects on the database are durable. As we do not - in this context - want to embark on a discussion of durability (and the related concept of persistence) we regard this property as having been fulfilled if changes to the object database made by committed transactions are recorded in the objects themselves.

We assume that an application system which is to be enhanced by the transaction components presented in this section consists of a number of Timor objects (which can have different types) that can be visited by processes/threads which make method calls on the objects. If the system to be enhanced is already a concurrent one, we assume further that its previous synchronisation requirements were satisfied by the use of dynamic synchronising qualifiers along the lines described in [4], and that when the system is transformed into a transaction system these are removed from the objects which they qualify by deleting them from the qualifier lists of target objects (as described in [5]). The transaction components which we now present can thus take over the responsibility for synchronisation (as part of their function of achieving atomicity, consistency and isolation) without side effects occurring from an earlier synchronisation strategy. (Since the actual synchronisation details of a transaction system depend on the strategy adopted, synchronisation will be assumed but not coded in this example.)

Architecture of the Transaction Components

The proposed transaction management system has the following major components:

- a single global transaction manager object,
- a separate qualifier for each database object, known as an object manager and
- threads in which the transactions are executed.

The global transaction manager object, of type `TrxManager`, implements decisions by the application to begin, commit and abort transactions. It maintains a global overview of the identities and progress of individual transactions as they access the application objects, and organises commit/abort decisions at the object level. This object is a "normal" Timor object, i.e. it is not a qualifier.

Each application object which can participate in transactions has its own associated object manager. This is a qualifier, of type `ObjectManager`, which has separate call-in bracket methods for controlling `op` and `enq` invocations of the target. These determine whether and how an individual transaction can access its associated target. They advise the global transaction manager whenever a thread touches its associated target object, and this then returns the identity of the transaction associated with the thread. In addition each

object manager has normal instance methods which the global transaction manager can invoke to advise it to commit/abort changes associated with its individual object.

Each transaction has a separate identity and executes in one or more threads of an application system. A thread remains associated with the same transaction until the latter completes, i.e. it can only be reused after the transaction with which it is associated either commits or aborts. At any given point in time there is a unique mapping from a thread number to a transaction number, but over the life of the system different transactions may be mapped to the same thread number(s).

This basic architecture is independent of the actual strategy adopted in a particular transaction management system (e.g. whether the approach is optimistic or pessimistic, whether before or after looks are used, etc.). It is not our aim to define a specific transaction system but to consider whether and how Timor can handle the basic mechanisms needed to implement such a system.

The Global Transaction Manager

The global transaction manager provides the application system with an interface via which transactions can be begun, committed and aborted. It also allows the object managers for the individual target objects to ascertain whether invocations of methods of the individual target objects are from registered transactions. There is one instance of this type per transaction system. It is used by a new *application* component which is responsible for scheduling transactions, known here as the *transaction scheduler*, which is not further described.

The main functions of an object of the type transaction manager (`TrxManager`) are to:

- allocate a transaction identifier for each new transaction,
- allow object managers to check whether a call to a target is from a registered transaction,
- determine whether a transaction can commit or must abort.

How it achieves these aims depends on the implementation strategy. Because in Timor a type can have multiple implementations, it is possible to envisage different implementations of the following types which embody different strategies.

Here is an outline type definition for a transaction manager:

```
type TrxManager {
instance:
 op TrxId beginTrx();
 /* The application's transaction scheduler invokes this
    operation when a transaction is initiated. It
    - allocates a unique trx id,
    - returns the trx id to the caller
 */
 op boolean commitTrx(TrxId trx) throws InvalidTransaction;
 /* This operation is invoked by the application transaction
```

```
      scheduler when a transaction completes. The TrxManager
      assesses whether the trx can commit (involving, for
      example, messages in the case of distributed
      transactions or validation in the case of optimistic
      transactions), and if so advises each object manager
      which has an object that has been touched by the
      transaction to commit.
      If the trx cannot commit (e.g. due to conflicts between
      transactions), the TrxManager aborts the trx.
      The return value indicates whether the transaction
      was committed (true) or aborted (false)
   */
   op void abortTrx(TrxId trxId) throws InvalidTransaction;
   /* This operation is invoked from the application
      transaction scheduler when a transaction has to abort.
      It advises each object manager which has an object that
      has been touched by the transaction to abort.
   */
   op TrxId trxNumber(AccessMode ac; Handle target;
                                        ObjectManager* objMgr);
   /* This method, called by object managers when
      their bracket methods detect an access to a target,
      - validates that the current thread belongs to a
        transaction,
      - notes for internal use the access mode parameter
        and the reference for the affected target object,
      - records the reference to the calling object manager
        for use later to advise the object manager to make a
        local commit or abort,
      - returns the trx id of the transaction (null = invalid)
   */
  }
```

Object Managers

The type `ObjectManager` is designed as a general qualifying type, i.e. it can qualify targets of different types in a general way, without a knowledge of the target type or its methods. When a writer method of a target is invoked, an `op` call-in bracket method of the associated object manager is activated. When a reader method is invoked, an `enq` call-in bracket method of the object manager is activated.

The main functions of the `ObjectManager` are:

- using bracket methods, to monitor reader and writer calls on the target object. This involves
  - checking whether calls on the target object belong to a registered transaction (by referring to the global transaction manager) and if not, rejecting them;
  - where appropriate making a shadow copy of the target for use by a transaction;
  - where appropriate redirecting calls to the appropriate shadow.

- when advised by the transaction manager via an instance method, either
  - to commit the changes for a transaction, or
  - to abort the changes

  and to remove any shadow.

The use of general bracket methods guarantees that different code does not have to be written to control different target types, and ensures that the same transaction management code can be re-used in different transaction systems.

Here is a type definition for an object manager.

```
type ObjectManager {
qualifies any:
 op bracket op(...) throws TransactionError;
 /* This bracket method catches writer calls. Inter alia it
    refers to the TrxManager to obtain the trx id of the
    current thread, advising the Transaction Manager that
    the access mode = WRITE.
    If the thread does not belong to a registered trx,
    the exception TransactionError is thrown
    Depending on the implementation strategy this bracket
    method may make a shadow copy of its object, to which
    application calls to the target may be redirected.
 */
 op bracket enq(...) throws TransactionError;
 /* This bracket method catches reader calls. Inter alia it
    refers to the TrxManager to obtain the trx id of the
    current thread, advising the Transaction Manager that
    the access mode = READ.
    If the thread does not belong to a registered trx,
    the exception TransactionError is thrown
    Depending on the implementation strategy this bracket
    method may make a shadow copy of its object, to which
    application calls to the target may be redirected.
 */
instance:
 op void localCommit(TrxId trxId);
 /* This instance method, called by the TrxManager, commits
    the trx from the viewpoint of its target object.
    This may involve
     - copying its shadow values to the target,
     - removing the shadow from a shadow set.
 */
 op void localAbort(TrxId trxId);
 /* This instance method, called by the TrxManager, aborts
    the trx from the viewpoint of its target object.
    This may involve, for example,
     - removing the shadow from the shadow set and/or
     - releasing locks held on the target object.
 */
```

```
maker:
 init(TrxManager* trxMan);
 /* The reference for the global transaction manager, allows
    it to call the method trxNumber.
 */
}
```

In implementations of the object manager there are certain activities which are especially relevant for the current investigation. They may need the ability:

a) unambiguously to identify both the current target object and the client's mode of access (read or write) to the target, in order to pass this information as parameters to the global transaction manager 's method

```
op TrxId trxNumber(AccessMode ac; Handle target;
                                    ObjectManager* objMgr);
```

b) to create shadow copies of a target object and to redirect the invocations of the controlled objects by transactions to these shadows.
c) to copy the content of a dynamically created shadow object back to the target (on a commit operation).

These objectives must be achieved in such a way that the code involved can be written without an awareness of the types involved. We consider these points in turn.

Identifying a Target and the Access Mode to the Target

Identifying the access mode to the target object (i.e. whether the method invocation is a reader or writer call) is easily distinguished in Timor. Method calls caught by the `op` bracket method are writers, while those caught by the `enq` bracket method are readers.

The type `ObjectManager` could have been designed with a maker which receives a handle for the target object. However, this would not guarantee that an actual reference passed in is actually for the target object. Instead the object manager can obtain a reference for its target object via a pseudo reference `target`, which of course has the type `TargetType`. In this sense the `body` statement of Timor, when used in isolation - as in earlier papers on qualifying types and bracket methods (e.g. [4, 5]) – can be viewed as a convenient short form for `target.body(...)`.

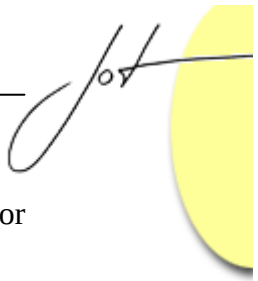
This mechanism enables object managers to invoke the global transaction manager from within the `enq` bracket method as:

```
trxMan.trxNumber(READ, target, this);
```

and from the `op` bracket method as:

```
trxMan.trxNumber(WRITE, target, this);
```

A protection issue similar to that mentioned in section 2 arises. In this case the programmer of the qualifying type could take advantage of the pseudo reference `target`



to access arbitrary methods of the target object and thus violate the integrity and/or confidentiality of the object. We return to this issue in section 4.

Creating Shadow Objects and Redirecting Method Calls to Them

In contrast with the example in section 2, a handle for a shadow object is not passed in via a maker. Instead the qualifier must dynamically create a shadow as a copy of the target object whenever one is needed. This can be achieved as follows:

```
state:
 Handle theShadow;
qualifies any:
 op bracket op(...) throws TransactionError {
  if (/* first access by the current transaction */)
   theShadow = new *target;
  ...
 }
```

Here a single shadow object is declared as a handle in the object manager's `state` section. (In implementations of some transaction strategies a set of shadows, e.g. one per transaction, might be necessary, but a single shadow is sufficient to illustrate the Timor features of relevance to our discussion.)

The statement used to instantiate an actual shadow object which is an exact copy of the target object (`theShadow = new *target;`) has the following semantics.

The dereferencing operator (`*`) copies the *value* of the target object, accessed via the special `target` reference mentioned above, and returns this value. The standard functionality of the `new` operator in Timor is that it accepts any value (here a copy of the target) as its operand, and transforms this into a new object, creating a reference for it. The assignment operator then copies this reference to the reference variable (in this case `theShadow`). Notice that static type checking is here not a problem, as `theShadow` is a handle, and can therefore accept a reference of any type, regardless of the actual type of the target and therefore of the shadow copied from the target.

As in the replica example in section 2, invocations of the target object's methods can be redirected to the shadow object using a cast statement:

```
cast (theShadow) as {
  (TargetType* myShadow) {
    try { return myShadow.body(); }
    catch (Exception e) {throw new AccessErr.init();}
  }
}
```

In appropriate cases the bracket method redirects the client's target method invocation to the shadow object (in this case without calling the target) and returns the result to the caller.

Copying the Content of a Shadow Object back to the Target

At this point it might be thought that all the key issues have been resolved. However, the qualifier must be in a position to copy the content of a shadow object back to the target object when a transaction commits. (The shadow cannot simply be used as a replacement for the target, as application components may have references to the original target.) There are two significant differences between this and the earlier copying of the content of a target object to a shadow. First the copying is to an existing object (i.e. the `new` statement is not relevant here). Second, the copy operation occurs in a normal instance method of the qualifier (e.g. the `commit` operation), not in a bracket method.

The second point is relevant because it means that neither the type `TargetType` nor the special reference `target` can be used, i.e. a statement such as

```
*target = *theShadow;
```

is not allowed in an instance method, because the pseudo reference `target` can only be used in bracket methods.

To circumvent this problem, the programmer can assign the target to a handle at the point where a shadow object is created, i.e. in the bracket method. Provided the handle is declared as a state variable it can then be used in normal instance methods, as the following code illustrates:

```
state:
 Handle theTarget, theShadow;
qualifies any:
 op bracket op(...) throws TransactionError {
  if (/* first access by the current transaction */)
   theTarget = target;
   theShadow = new *target;
  ...
 }
```

At this point the state of the Object Manager contains a reference pointing to the target object and a further reference pointing to the new shadow object, the content of which is a copy of the target object.

In the `localCommit` instance method a statement such as

```
*theTarget = *theShadow;
```

is now in principle possible, but this must appear in a context where the compiler can check the types, i.e. in a cast statement. However `TargetType` cannot be used outside bracket methods because general purpose qualifiers can (even concurrently) qualify targets of different types (see [5]), so that outside a bracket method `TargetType` is ambiguous.

However, a different Timor concept can be used to solve this problem. The crucial issue is not that the compiler needs to know both types involved, but rather that it can be sure they have the *same* type. To resolve such problems Timor supports a special type called `SameAs`, for use in the declaration part of cast clauses. This allows the compiler to

check that the dynamic types of the subject variable and that of another variable are identical. During the execution of the corresponding cast clause not only the subject reference (here `thisShadow`) but also the reference with which it is to be compared must be `fixed`, i.e. new assignments cannot be made to them (although the contents of the objects to which they refer can be changed). As mentioned earlier, an auxiliary reference used within a cast declaration (here `thisShadow`) is always implicitly defined to be `fixed` (for a reference). A separate auxiliary reference can be used to fix the target object reference, as follows:

```
fixed Handle thisTarget = theTarget;
cast (theShadow) as {
 (SameAs thisTarget thisShadow) {*thisTarget = *thisShadow;}
else throw new TransactionError.init();
}
```

If the programmer were to attempt to make an assignment such as `*theTarget = *theShadow` either inside or outside the cast statement a compile time type error would occur, because both are just handles.

Normally cast statement clauses are matched polymorphically with the variable which is the subject of the cast (e.g. a `(Person p) {...}` clause would be selected where the subject of the cast is a `Student` reference, assuming that `Student` is a subtype of `Person`). However, the `SameAs` clause makes an exact type check, giving the programmer the opportunity to use cast statements to check dynamic types exactly.

4 PROTECTING TARGETS FROM QUALIFIERS

In this section we consider how Timor can be used to prevent qualifiers from breaching the security of their targets (and any shadow objects which are copies of the targets). As a preliminary we briefly describe relevant aspects of Timor's view construct.

Views

A view defines a group of related instance methods which can typically be incorporated into many types, and its primary purpose is to encourage the idea of programming to interfaces. Here is a simple example:

```
enum Openmode {CLOSED, READ, WRITE}
view Openable {
 op void open(Openmode mode) throws OpenError;
 op void close() throws NotOpen;
 enq Openmode openMode(); // returns current open mode
}
```

A reference variable can have a view as its type, thus allowing objects of different types which contain the view methods to be used polymorphically. Hence objects of any "openable" type can be assigned to a reference

```
Openable* openRef;
```

and the `Openable` methods can be invoked via it.

Restricting Access to Objects

The pseudo reference `target`, as described in section 3, apparently gives any bracket method the right to invoke methods of its target object. This would potentially provide a mechanism by which programmers could violate both the integrity of the target (by invoking its `op` methods) and its confidentiality (by invoking its `enq` methods). The same threats also arise through the passing in of a replica via a handle in the replica writer example in section 2.

Timor supports a simple mechanism which the programmer can use to prevent such breaches. Any reference variable declaration, including that for a handle, can be restricted in a manner analogous to the mechanisms for restricting access via capabilities in operating systems. This is achieved syntactically by listing in special brackets (`[:` and `:])` a set of access permissions after the type of the reference. Each permission can be expressed as a view.

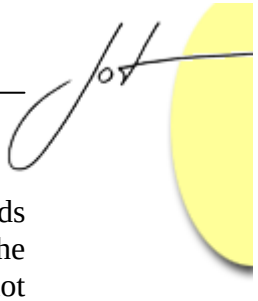
Given the above definition of `Openable` and an object of type `File`, which includes the above methods `open`, `close` and `openMode` (regardless whether they were explicitly incorporated as a view in the original type definition or were defined as normal methods of the type), access via the following reference variable `fileRef` would allow only these methods to be accessed.

```
File*[:Openable:] fileRef;
```

Such a reference is called a *restricted reference*. Downcasts cannot be performed on restricted references.

The difference between `fileRef` and `openRef` is that any object of any type which incorporates the view `Openable` can be assigned to `openRef`, as for normal subtyping, whereas objects only of the type `File` (and its behavioural subtypes) can be assigned to `fileRef` and only the methods of `Openable` can be called via this reference. Hence this mechanism is more restrictive than normal subtyping.

The set of permissions are defined using the same set syntax as is used for defining subsets in the Timor Collection Library. The operators are `+` (union), `-` (difference), `*` (intersection).



The rule for assigning objects to restricted references defines that the methods invocable from the restricted reference must be a subset of those invocable via the reference from which the assignment is made. Hence access can be reduced but not increased by using different variables or by passing parameters.

The relevance of this mechanism to the problem at hand is that there are also certain predefined views, including `op`, `enq`, and `body`. These restrict the use of a reference variable to allowing only calls to writer methods, to reader methods, or to a single invocation of `body` (only within a bracket method).

Target References and the Body Permission

To solve the problems outlined above with respect to the replica writer example, the `body` permission can be used as follows:

```
state:
 Handle[:body:] theReplica;
maker:
 init(Handle[:body:] aReplica) {theReplica = aReplica;}
qualifies any:
enq bracket op (...)
                throws DynamicTypeErr, ReplicaAccessErr {
 cast (theReplica) as {
  (TargetType[:body:]* myReplica) {
    try { myReplica.body(...); }
    catch (Exception e) {throw new ReplicaAccessErr.init();}
  }
  else throw new DynamicTypeErr.init();
 }
 return body(...);  // call the database object
}
```

Here the reference for passing in the (already existing) replica is restricted to one invocation of `body(...)` per activation of a bracket method. Other methods cannot be called at all. The restriction that `body(...)` can be invoked once only in the activation of a bracket method is a general restriction, i.e. although a `body` statement for a particular target can appear statically at different points in a bracket method, only one dynamic execution is permitted. This is important to ensure that repeated activations of `body(...)` cannot be used to violate the integrity of the target object in the form of a replay attack.

In the replica writer example the `body` permission is visible to the client who creates a qualifier, via the maker parameter. Since this guarantees that the object which he passes in can subsequently be further assigned only to references also restricted to a single body invocation, the client (who associates the qualifier with his target object) can be sure that no other methods of the target are invoked by the qualifier and that for any bracket method activation only a single invocation of `body(...)` is permitted.

Restrictions on the Use of Target

The transaction processing example introduces the pseudo reference `target`. It is also essential that the instance methods of the target object cannot be invoked via this pseudo reference. This is guaranteed by the fact that `target` is itself restricted to a `body` permission, i.e. its implicit definition is:

```
TargetType[:body:]* target;
```

Copying Objects

The remaining issues in the transaction processing example are concerned with misusing the shadow. Shadow objects are not passed in as parameters and therefore cannot be restricted by clients passing in a restricted reference, in contrast with the replica example. The key issue here is that a shadow object is created by copying the content of `target`, i.e.

```
theShadow = new *target;
```

The solution is therefore self evident. The rule that an object can only be assigned to a reference which is at least as restricted as the reference which is the source of the assignment is extended to apply to copies of objects. In other words, because `target` is accessed via a reference with only a `body` permission, a copy can only be assigned to a references which has at most the `body` reference permission. Hence the above assignment will fail unless `theShadow` is declared as:

```
state:
 Handle[:body:] theShadow;
```

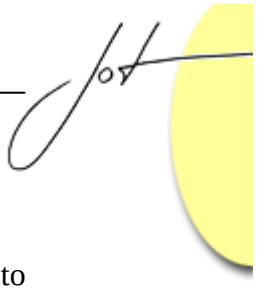
Corresponding changes must then also be made to the code. In the bracket methods the following modification is required for the cast statement:

```
cast (theShadow) as {
  (TargetType[:body:]* myShadow) {
    try { return myShadow.body(...); }
    catch (Exception e) {throw new AccessErr.init();}
}
```

In the local commit method the statement

```
cast (theShadow) as {
 (SameAs theTarget thisShadow) {*theTarget = *thisShadow;}
}
else throw new TransactionError.init();
```

remains unchanged, since `SameAs` checks the exact type.



Confining Objects

Although the measures described earlier in this section are thought to be sufficient to prevent the flow of information from a target via a qualifier to an unauthorised third party, the owner of a target, when he creates the transaction qualifiers, can qualify these with further qualifiers which have call-out brackets designed to confine information which the transaction qualifiers attempt to release to other components [6].

Such brackets could prevent invocations of any methods except calls from an object manager to the global transaction manager's `trxNumber` method. This releases references for both the object manager itself and for its target. However, the target reference is needed only for identification purposes and should never be called by the transaction manager. This can be guaranteed by defining `trxNumber` to have a restricted reference as its parameter, i.e. `Handle[: :] target`. This empty set restriction indicates that no methods can be called via this reference.

The reference for the object manager, on the other hand, is needed by the transaction manager to invoke the former's commit and abort methods and so cannot be completely restricted. In this case the restriction must be to a view which permits these calls, e.g.

```
view TrxFinalisation {
 op void localCommit(TrxId trxId);
 op void localAbort(TrxId trxId);
}
```

These restrictions are reflected in the following redefinition:

```
op TrxId trxNumber(AccessMode ac; Handle[: :] target;
                   ObjectManager[:TrxFinalisation:]* objMgr);
```

Copying Objects with Qualifiers

When an object is copied, as in the statement

```
theShadow = new *theObject;
```

the *dynamic* qualifiers in the qualifier list [5] of the original object are not automatically associated with the new object, neither in their original form nor as copies. However, dynamic qualifiers can be associated with a new object as it is created, e.g. in a literal qualifier list or they can even be added later, if a named list is used. For example the above statement might be modified to read:

```
theShadow = new {qualifier1, qualifier2} *theObject;
```

In this case two qualifiers (which might or might not also be in the list of the original object), are associated with the new object.

If an object is defined (at the type level) to have *static* qualifiers [7] these are considered to be an integral part of the object and are automatically copied as part of an object which is copied, i.e. they are part of the value returned by the dereferencing operator `*`.

Preventing the Copying of Objects

The use of the dereferencing operator to copy objects can itself in principle present a threat to the confidentiality of an object. Suppose for example that an object is protected by a qualifier which contains an access control list (ACL) [9], it might appear possible to circumvent this protection by copying the object and then accessing its information as stored in the copy.

However, this situation can be prevented by the use of bracket methods. When a dereferencing operator occurs on the left side of an assignment statement it indicates that the object's value will be overwritten by the value of some other object of the same type. Just as general bracket methods can be defined as `op` and `enq` brackets (to monitor operations and enquiries), so also a bracket method `overwrite` can be defined which monitors overwriting (and if appropriate prevents this by not calling the `body` statement). Similarly a `copy` bracket method can be defined to monitor and potentially prevent the use of dereferencing on the right side of an assignment statement or more generally in an expression. If `overwrite` and/or `copy` bracket methods are not defined, then `op` and/or `enq` brackets (if they exist) are applied to dereferencing operations.

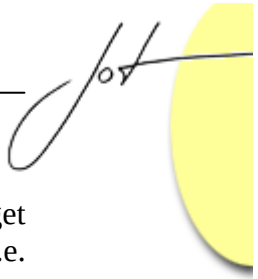
Preventing the overwriting or copying of objects can also be achieved by references which restrict the use of these operations (e.g. `Handle[:all-overwrite:]`).

5 RELATED WORK

To our knowledge the only other work which is closely related to the programming language concepts described in the present paper is that being carried out in the aspect oriented programming (AOP) community.

Timor's qualifying types can be seen as a technique for defining and implementing aspects, but in contrast with languages such as AspectJ [8] and AspectC++ [11] it supports various features which make it particularly suitable for supporting transactions and similar "system oriented" activities. These include the ability:

- to distinguish between reader and writer methods without needing to resort to special naming conventions and pattern matching,
- flexibly to associate qualifiers with individual objects rather than associating them statically with a class,
- dynamically to add qualifiers to and remove them from existing objects [5],
- to associate the same qualifier with a group of objects rather than with a single object or with all objects of a class,



- to define aspects without a knowledge of, or a need to modify the code of, a target (at any level, whether in the source, intermediate code or compiled code), i.e. without the need to define "pointcuts",
- to define aspects without a knowledge of each other, even in cases where precedence is considered to be relevant,
- to define general qualifying types and specialised types based on view interfaces without a knowledge of (nor the presence at compile time of) each other's source or bytecode or that of types which they might qualify,
- to introduce new methods ("introduction" in AOP terminology) without these becoming methods of the qualified objects (important for example where the qualifier controls protection, so that the client cannot change the protection conditions).

Recent AOP developments, the so called "second generation" AOP projects, e.g. [1, 3], have addressed some of these issues. In particular criticisms of the static approach of AspectJ have led to a more dynamic approach, as exemplified by JBoss [3]. This frees aspects from being bound to types, allowing advice to be dynamically associated with and removed from individual objects. This is achieved by binding advice to method calls using XML. In this way, methods can be either directly bound to aspect advice or can be "prepared" for such advice (the more dynamic case), and advice can be instantiated and then attached to advised objects as appropriate. While this approach is certainly more flexible than AspectJ through its more dynamic features, it still resorts to the use of a second language level (XML) to associate aspects with objects. This is fundamentally different from Timor, where "aspects" (qualifiers) are completely defined in terms of the normal features of the languages and can (with complete flexibility) be added to and removed from individual objects simply by manipulating a normal Timor list associated with the object, which is itself a first class element of the language.

Finally middleware has been claimed to be the "killer application" for AOP:

"Much application-server functionality can be cleanly and logically expressed as aspects. Context passing, remoting, security and transactions can be thought of as add-on functionality that happens 'around' (before and/or after) a method call to an ordinary object. Aspect-oriented programming allows an application server designer to provide these features without requiring service developers to extend abstract classes or implement interfaces" [10].

In the present paper we have demonstrated how transactions can be supported in Timor using qualifying types (the Timor equivalent to aspects). In earlier papers we have also shown how qualifying types can play a significant role in improving security. But we are not convinced that they will in future have a significant role to play in "remoting" (remote procedure calls), or in implementing persistence (which is added to the list in another part of the same paper). In the context of Java this claim makes sense because remote procedure calls and persistence are not fundamental features of the Java language, and so must be treated as add-ons. But Timor, as a new language, presupposes a fundamentally different model, in which both remote procedure calls and persistence are basic architectural concepts. Hence they do not need to be provided as add-ons, and

consequently qualifiers have no role to play in their implementation. This is the subject of a paper currently in preparation.

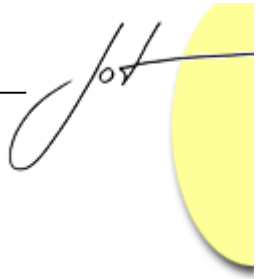
Finally, it is not clear to us how the AOP approach can provide a level of protection approaching that described in section 4, whereby the integrity and confidentiality of an application can be safeguarded from components developed to provide a transaction processing environment (or other separately coded aspects).

6 CONCLUSION

The paper has demonstrated how Timor can employ qualifying types to handle a certain class of "system" issues, including the maintenance of replicas and support for transactions in the traditional database sense. It has also demonstrated how target objects can be protected from wayward qualifiers. Perhaps its most interesting contributions in terms of the technicalities of qualifying types are (a) the idea that the language provides (restricted) access both to the target object itself (using the keyword `target`) and to the type of the target (using the keyword `TargetType`), and (b) that invocations of a target can be deflected by bracket methods to replicas/shadow objects.

The effect of these features is that a well-structured Timor application system which was not originally designed to work in transaction mode can be updated to a transaction system by introducing autonomous components. These new components can be designed, developed and implemented completely independently of individual application systems and without changing the code (at any level) of the application objects on which transactions are carried out. Thus a generic set of transaction components can be developed and then applied to a wide range of disparate existing applications. Furthermore different transaction strategies (e.g. optimistic, pessimistic) can be supplied in alternative implementations of the component types.

Finally transactions have been used in this paper as a demonstration of the flexibility just described. But this flexibility can equally easily be applied to the retrofitting of other kinds of requirements, such as security, monitoring, protection and so on.



REFERENCES

[1] CollabNet, "dynaop," https://dynaop.dev.java.net/, 2004.

[2] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Fransisco. Morgan Kaufmann, 1993.

[3] JBoss, "JBoss Aspect Oriented Programming," http://www.jboss.org/developers/projects/jboss/aop, 2004

[4] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe2002, Erfurt Germany*, vol. LNCS 2591, M. Aksit, M. Mezini and R. Unland, Eds.:Springer 2003, pp. 330-344, http://link.springer.de/link/service/series/0558/papers/2591/25910330.pdf

[5] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology*, vol. 3, no. 1, Jan.-Feb 2004, pp. 101-121, http://www.jot.fm/issues/issue_2004_01/article1.

[6] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Call-out Bracket Methods in Timor," vol. 5, no. 1, January-February 2006, pp 51-67, http://www.jot.fm/issues/issue_2006_01/article1.

[7] J. L. Keedy, K. Espenlaub, G. Menger, C. Heinlein, and M. Evered, "Statically Qualified Types in Timor," *Journal of Object Technology*, vol. 4, no. 7, 2005, pp. 115-137, http://www.jot.fm/issues/issue_2005_09/article5.

[8] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," ECOOP 2001 - Object-Oriented Programming, 2001, Springer Verlag, LNCS, vol. 2072, pp. 327-353.

[9] B. W. Lampson, "Protection," Proc. 5th Princeton Symposium on Information Sciences and Systems, 1971

[10] D. Schweisguth, "Second-generation aspect-oriented programming," *Javaworld*, http://www.javaworld.com/javaworld/jw-07-2004/jw-0705-aop-p3.html, July, 2004.

[11] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 53 - 60.

About the authors

J. Leslie Keedy recently retired from the position of Professor and Head, Department of Computer Structures, University of Ulm, Germany, where he lead the Timor language design and the Speedos operating system design groups. His email address is keedy@jlkeedy.net. His biography can be visited at http://www.jlkeedy.net/biography_short.php

Klaus Espenlaub completed his Ph.D. in Computer Science at the University of Ulm in 2005. Currently he works as a research assistant in the Department of Computer Structures at the University of Ulm. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is klaus@espenlaub.com.

Christian Heinlein received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently, he works as a scientific assistant in the Department of Computer Structures at the University of Ulm. His research interests include programming language design in general, especially genericity, extensibility and non-standard type systems. His email address is christian.heinlein@uni-ulm.de.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently she works as a scientific assistant in the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering. Her email address is gisela.menger@uni-ulm.de.

Frans Henskens is Assistant Dean of the Faculty of Engineering & Built Environment at the University of Newcastle, Australia. His research interests centre on engineering of flexible software systems, bioinformatics, computational neuroscience, distribution using global virtual memory, programming language design, resilience and availability in database systems and use of persistent stores for bulk data storage and manipulation. His email address is Frans.Henskens@newcastle.edu.au

Michael Hannaford is a Senior Lecturer in Computer Science and Software Engineering at the University of Newcastle, Australia. His research interests centre on virtual memory management, distributed systems, programming language design, and object-oriented software engineering. His email address is Mike.Hannaford@newcastle.edu.au